Instant web applications with Meteor

Build responsive web applications, from concept to scaled deployment, in record time

Sing Li (westmakaha@yahoo.com)
Consultant
Makawave

Skill Level: Advanced

Date: 14 Jun 2013

With Meteor, a new web application-creation platform, JavaScript developers can build highly interactive and responsive rich-client web applications simply and rapidly. Meteor proposes a new way to think about web application design and development, with the aim of simplifying and dramatically shortening the development cycle. Sing Li goes beyond the hype and explores the promise of Meteor with two functional, nontrivial application examples — one web-based and one mobile — and coverage of the Meteor architecture. You'll gain hands-on experience building Meteor applications while leveraging popular-industry standard JavaScript libraries.

Meteor, a new platform for web application development, is gaining significant international adoption. More than a JavaScript coding framework, Meteor provides an innovative way to construct scalable, rich, interactive web applications. Meteor promises to turbocharge the development cycle by simplifying the coding model and reducing the amount of code that developers must write. Using Meteor, experienced web application architects and developers can go from concept to full deployment in days or weeks, instead of the customary months or longer.

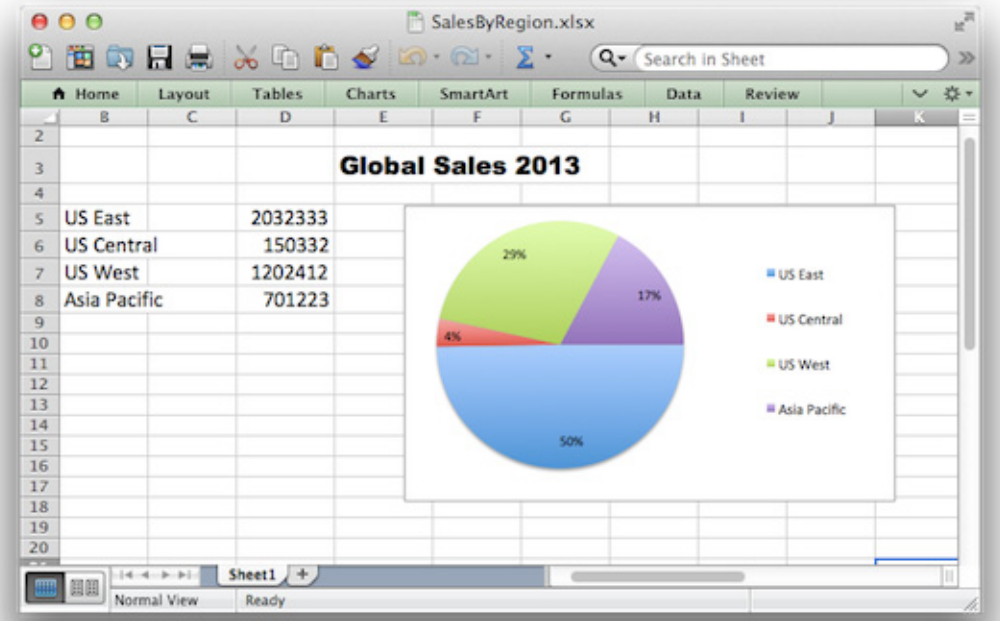
For a step-by-step guide on installing and starting to develop with the Meteor platform, a great resource is the developerWorks article "Develop easy real-time websites with Meteor." In this current article, I take you deeper into Meteor development with two detailed application examples, and give you an overview of the Meteor architecture. With this knowledge, you'll be able to decide for yourself if rapid web application creation on Meteor is right for you.

Looking back to the future

The approach that Meteor proposes is revolutionary in a sense, yet it has some evolutionary aspects too. It continues along the same IT path as one of the great

successes in computing history: spreadsheet software. Figure 1 shows a typical spreadsheet example — a Sales by Region spreadsheet with a pie chart:

Figure 1. The Sales by Regions spreadsheet



Trying out the common spreadsheet

You can try out the spreadsheet in Figure 1, which is part of the article's sample code download. Change any of the sales figures, and you see the pie chart update automatically.

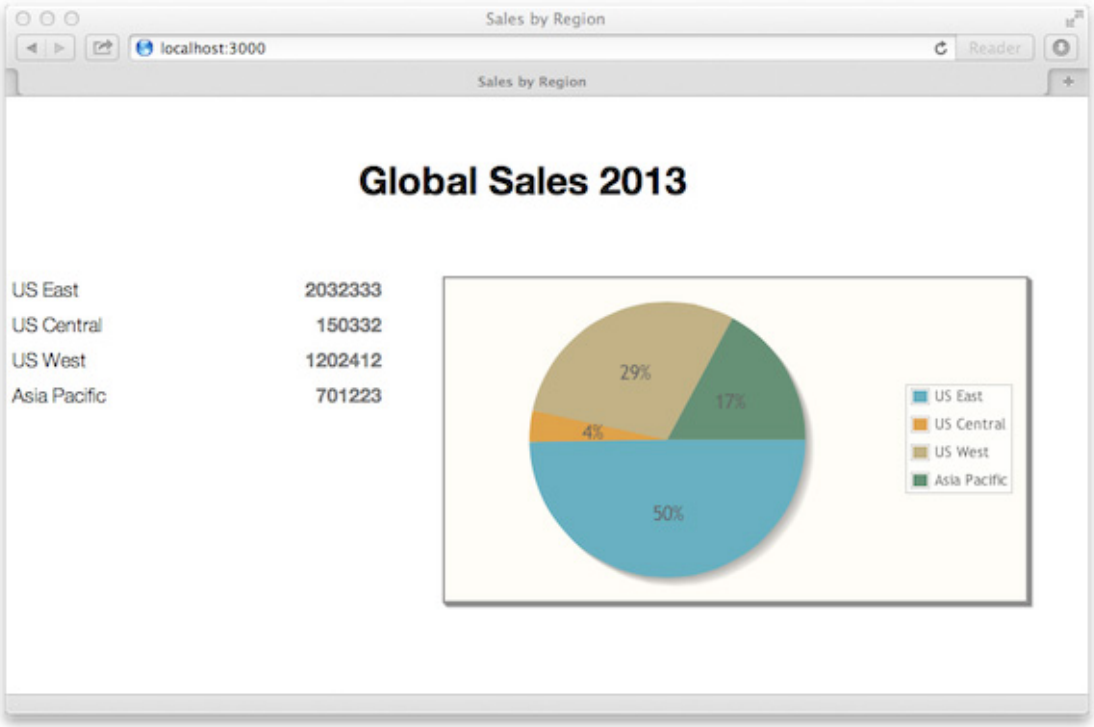
If you modify any of the regional sales figures in the Sales by Region spreadsheet, the total sales figure (not shown) changes and the pie chart instantaneously redraws to reflect the new relative proportions of the slices.

Nowadays this is neither novel nor interesting. But imagine yourself back in 1983 when Lotus 1-2-3 unveiled these features to early PC users, and you might have an appreciation of its impact. Never before could one do so much work with so little programming. Although spreadsheet software wasn't initially intuitive, most users could operate it proficiently in a matter of days. Spreadsheet software is still one of the killer apps driving PC sales worldwide.

Fast-forward three decades

Thirty years after the first spreadsheet software came out, you can see how the spreadsheet metaphor has evolved with Meteor. Figure 2 shows the Sales Portal web application, created with Meteor circa 2013:

Figure 2. Sales Portal web application



Sales Portal shows up-to-the-minute regional sales figures and a corresponding pie chart. As the hypothetical CEO, you can monitor the sales figures, and each of your regional sales teams can update its sales figure periodically.

You can download the Sales Portal application and go hands-on with it if you have Meteor installed. Change to the download's sales_nologin subdirectory and type `meteor run`. Point a browser instance at http://localhost:3000/, and the regional sales figures and pie chart should appear. Double-click any sales figure to change it. The pie chart updates immediately after you confirm your change. If you point multiple browser instances at the Sales Portal, all of them update to show the latest sales figures, and you can modify the figures from any one of the browser instances. (If you're unable to install Meteor, you can try the hosted versions of the applications; see Resources.)

Figure 3 shows the US Central team selecting and updating its sales figure:

Figure 3. Updating US Central sales

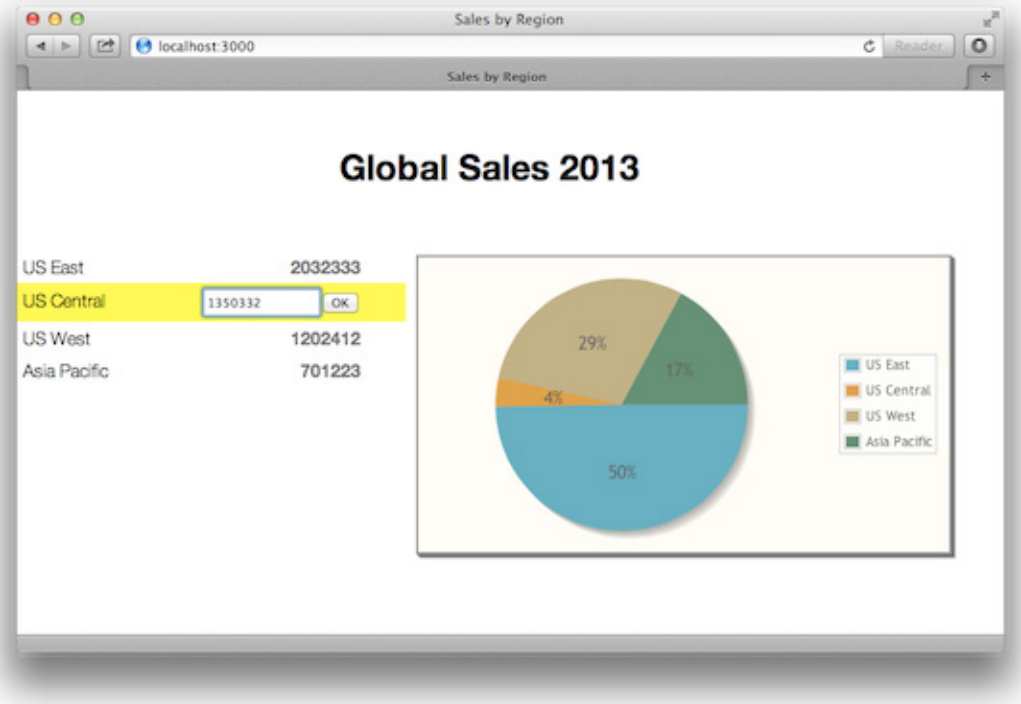
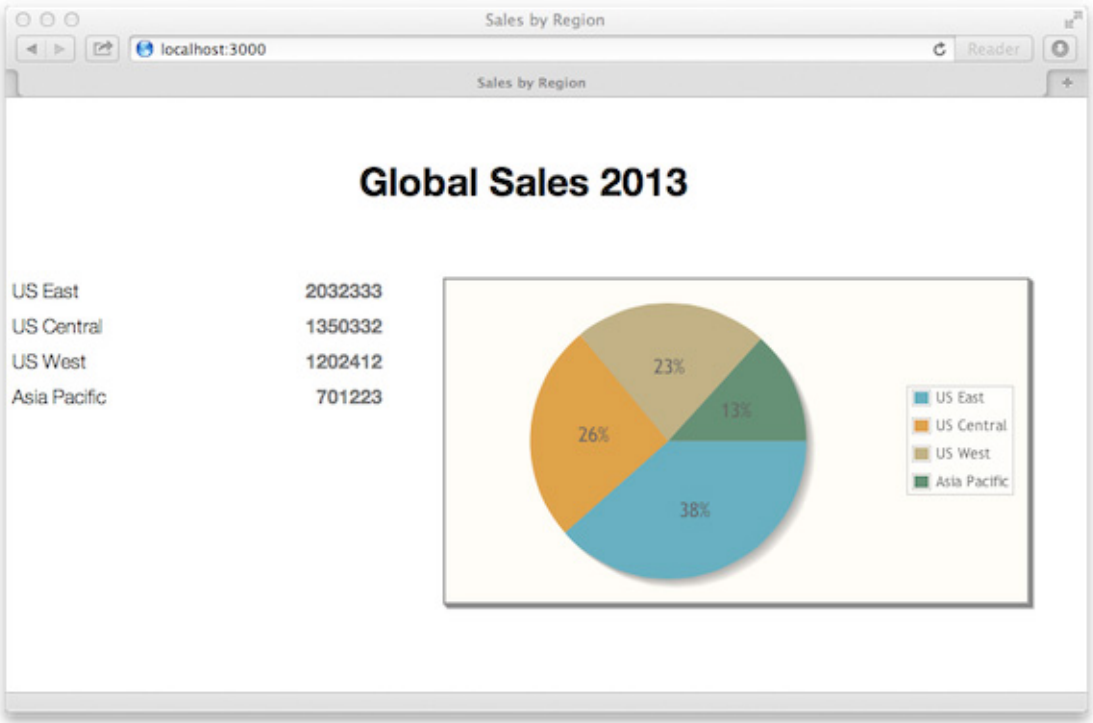


Figure 4 shows the updated US Central sales figure and the up-to-date pie chart. Any user concurrently accessing the Sales Portal at the time of the update sees the change immediately.

Figure 4. Pie-chart proportions updated to reflect new US Central sales figure



Instead of manual updates, you can also imagine a back end where the sales figures consist of subaggregates that are gathered and consolidated autonomously before updating. The visual presentation of the Sales Portal application is identical to that of the venerable spreadsheet, but Sales Portal now also has:

- Global Internet access via the ubiquitous browser
- Simultaneous access by multiple users
- Optional automated back-end data aggregation and consolidation

Significant effort would be required if you were to design, code, and deploy such a system using standard enterprise technology (say, a Java™-based tool chain). Meteor minimizes this effort dramatically, as you see in the code walk-through that follows.

Reactive thinking

Meteor reactive defaults

Meteor has certain sources of data that are reactive by default. These currently include:

- Meteor collections — typically, the results of MongoDB queries
- Variables explicitly bound to Meteor's `Session` singleton
- `Meteor.user`, `Meteor.userId`, and `Meteor.logginIn`, which track current user and login status
- `Meteor.status`, which tracks server connection status

One key feature of the spreadsheet is its reactive nature. In the Sales by Region example, when a regional sales figure is updated, everything else that depends on

that figure is recalculated on the fly. If the dependent component renders graphical output, such as the pie chart, the chart is immediately redrawn with updated slice sizes. You needn't write the code that manages the dependencies (which can be complex) or the code to update the components such as the pie chart. All you do is declare the reactive elements (the sales figures) and their dependencies (total sales and pie chart in this case). The spreadsheet takes care of it all.

Now imagine doing the same today with a web application, and you've got a good picture of how Meteor proposes to simplify web-based system creation.

When you design a Meteor application, you decide on the reactive elements — say, the regional sales data collection. Then, you lay out your presentation layer using standard HTML, CSS, client-side JavaScript libraries and components (such as JQuery, JQuery UI, or Underscore), and templating technology such as Handlebars (similar in concept to JavaServer Pages, typically runs on the client side; see Resources). Meteor tracks all dependencies of the reactive elements and then rerenders visual elements and recomputes dependencies to reflect the latest updated values.

This approach greatly reduces the amount of infrastructure code that you need to write, debug, and test. You don't need to write the custom back-end web services to sync the update request, the code to update the database or datastore, the code to push change notifications out to other connected clients, or the code to fetch updated values from the back end upon notification.

Deep dive into the Sales Portal code

Listing 1 shows the sales.js file, which contains all the server- and client-side logic behind the Sales Portal application. This is the only JavaScript code that I needed to write for this application. (You can find sales.js in the sales_nologin directory of the code download.)

Listing 1. Client- and server-side logic for Sales Portal: sales.js

```
Sales2013 = new Meteor.Collection("regional_sales");

if (Meteor.is_client) {
  Template.salesdata.dataset = function () {
    return Sales2013.find({});
  };

  Template.datapoint.selected = function () {
    return Session.equals("selected_datapoint", this._id) ? "selected" : '';
  };

  Template.datapoint.events = {
    'click': function () {
      Session.set("selected_datapoint", this._id);

    }
  };

Template.salesdata.rendered = function()
{
```

```
 $('.editable').editable(function(value, settings) {
    Sales2013.update(Session.get("selected_datapoint"),
{$set: {total: parseInt(value)}});
    return(value);
 }, {
    type    : 'text',
    style : 'inherit',
    width : 100,
    submit  : 'OK',
});

    var cur = Sales2013.find();
    if (cur.count() === 0)  // do not render pie if no data
      return;
    var data = [];
    cur.forEach( function(sale) {
      data.push( [sale.region, sale.total]);
    });
  var plot1 = $.jqplot ('chart', [data],
    {
      seriesDefaults: {
// Make this a pie chart.
renderer: $.jqplot.PieRenderer,
rendererOptions: {
  // Put data labels on the pie slices.
  // By default, labels show the percentage of the slice.
  showDataLabels: true
}
      },
      legend: { show:true, location: 'e' }
    }
  );
}

}

if (Meteor.is_server) {
  Meteor.startup(function () {
      Sales2013.remove({});
      Sales2013.insert({region:"US East", total: 2032333});
      Sales2013.insert({region:"US Central", total: 150332});
      Sales2013.insert({region:"US West", total: 1202412});
      Sales2013.insert({region:"Asia Pacific", total: 701223});
  });
}
```

Observe the conditionals in Listing 1 around the `Meteor.is_client` and `Meteor.is_server` variables. These are runtime context indicators that are provided by the Meteor core and usable anywhere in your code. In this case, they allow the combination of client- and server-side code within the same source.js file. Any code outside of the conditionals runs on both the client and server.

Alternatively, you can completely separate the client and server source code by placing client-side code in a subdirectory named client and server-side code in a subdirectory named server. In that scenario you put any common assets that are required by both client and server in a subdirectory named public. A more secure version of the Sales Portal application, which you'll read about later in this article, uses that directory structure.

Identifying the reactive data

Another reactive data source

The `selected_datapoint` session variable in Listing 1 is also reactive. (See the Meteor reactive defaults sidebar for more information about elements that are reactive by default.) It's used in this case to change the sales figures' row highlighting. Row highlighting is performed via changes in dynamic CSS styles. The `selected_datapoint` session variable is updated when a user clicks a row. Because Meteor rerenders dependencies every time this variable changes, the highlight is updated accordingly.

One of the reactive sources of data for the Sales Portal application is a query on the `Sales2013` Meteor collection. You can see its use in this client-side code fragment from Listing 1:

```
Template.salesdata.dataset = function () {
   return Sales2013.find({});
};
```

Because the query is reactive, all of its dependencies are recomputed or rerendered when the query result set changes. This is how the sales figures and the pie chart, across all browser instances, get updated. Listing 2 shows the associated HTML template code, found in the sales.html file in the sales_nologin directory of the code download:

Listing 2. Client side HTML and templates: sales.html

```
<head>

<title>Sales by Region</title>
</head>

<body>
  <div id="title">
    <h1>Global Sales 2013</h1>
  </div>
  <div id="container">

    <div id="salestable">
      {{> salesdata}}
    </div>
    <div id="chart">
    </div>
  </div>

</body>



<template name="salesdata">
  <div class="salesdata">
    {{#each dataset}}
      {{> datapoint}}
    {{/each}}
  </div>
</template>
```

```
<template name="datapoint">
  <div class="datapoint {{selected}}">
    <span class="region">{{region}}</span>
    <span class="sales editable">{{total}}</span>
  </div>
</template>
```

The HTML file in Listing 2 is a Handlebars template, which Meteor currently supports. You can see the Handlebars expressions in `{{ }}`. Through its Spark engine (described in this article's An architecture for modern web applications section), Meteor can work with other JavaScript templating components.

The rows of sales figures are rendered via the `salesdata` template code, highlighted in bold in Listing 2. Because this template depends on the `dataset` helper function, shown in Listing 1, it is rerendered every time the contained reactive query changes.

Seeding sample data on the server

The initial regional sales data for Sales Portal is seeded by the server-side code (from Listing 1) that's shown in Listing 3:

Listing 3. Server-side code to seed data in MongoDB

```
if (Meteor.is_server) {
  Meteor.startup(function () {
      Sales2013.remove({});
      Sales2013.insert({region:"US East", total: 2032333});
      Sales2013.insert({region:"US Central", total: 150332});
      Sales2013.insert({region:"US West", total: 1202412});
      Sales2013.insert({region:"Asia Pacific", total: 701223});
  });
}
```

Meteor's latency compensation

Meteor has a feature called *latency compensation*. Latency compensation is basically a visual rendition of the *eventual consistency* concept in big data management. When you update data on the client via the Minimongo stub, any changes are reflected immediately on the client — including reactive rerendering. The changes are also propagated to the server. However, the propagated changes might not succeed, for various reasons — including denied access. The publish-subscribe mechanism takes care of making sure that the client eventually (usually very shortly) reflects the actual state of the server. Latency compensation enables a highly responsive UI that is holdup-free, a hallmark trait of modern Web 2.0 applications. The cost is potentially a short moment of visual data inconsistency.

On the Meteor server, a full MongoDB instance is in operation. This full instance can accept queries and updates from clients other than Meteor.

On the Meteor client, the same JavaScript MongoDB API is also available. This unifies the client and server coding and enables code reuse on both client and server.

The client-side API is provided by a smart stub called Minimongo. Minimongo uses latency compensation to reflect database changes. Because Minimongo typically deals with small client-side datasets, it doesn't support indexing.

A publish-subscribe model is used to control the data that's synchronized between the MongoDB server and the Minimongo client. By default, all server-side Meteor collections are published. Meteor uses Distributed Data Protocol (DDP) to move data between client and server. (DDP drivers in the form of stub and provider can be created for other databases; ongoing efforts in the Meteor community include an upcoming MySQL driver.)

Integrating jQuery plugins

Sales Portal uses the jqPlot jQuery plugin to render the pie chart. The rendering and rerendering of the pie chart is driven reactively by data changes in the `Sales2013` collection. You saw earlier that the `salesdata` template is rerendered every time the `Sales2013` collection changes. Listing 4 shows the client-side function (from Listing 1) that redraws the pie chart when triggered by the `salesdata` template's `rendered` event:

Listing 4. jQuery code to render the pie chart using the jqPlot plugin

```
Template.salesdata.rendered = function()
{

 $('.editable').editable(function(value, settings) {
    Sales2013.update(Session.get("selected_datapoint"),
{$set: {total: parseInt(value)}});
    return(value);
 }, {
    type    : 'text',
    style : 'inherit',
    width : 100,
    submit  : 'OK',
});


    var cur = Sales2013.find();

    if (cur.count() === 0)  // do not render pie if no data
      return;
    var data = [];
    cur.forEach( function(sale) {
      data.push( [sale.region, sale.total]);
    });
 var plot1 = $.jqplot ('chart', [data],
   {
     seriesDefaults: {
// Make this a pie chart.
renderer: $.jqplot.PieRenderer,
rendererOptions: {
  // Put data labels on the pie slices.
  // By default, labels show the percentage of the slice.
  showDataLabels: true
}
     },
     legend: { show:true, location: 'e' }
   }
```

```
  );
}
```

Sales Portal uses the Jeditable jQuery plugin to enable in-place editing of the sales figures. The code that handles editing appears between the `Template.salesdata.rendered = function()` and before `var cur = Sales2013.find();` lines in Listing 4.

See Resources for more information on jQuery and the jqPlot and Jeditable plugins.

Understanding Meteor's stylesheets' and scripts' loading order

For the jQuery plugins to be loaded successfully, it's important that their associated CSS files and JavaScript code be loaded in the proper order.

Notice in Listing 2 that the sales.html file contains no `<script>` tags or `<link type="text/css" ... >` stylesheet. Instead, Meteor loads client-side scripts and stylesheets automatically by scanning directories: the deepest directory first, then in alphabetical order within each directory.

To take advantage of this loading order (and the fact that the name of the script or CSS file doesn't matter) I renamed some scripts to ensure their loading position. For example, jquery.jeditable.mini.js was renamed to zjquery.jeditable.mini.js under the client/js directory to make sure it loads last. I also renamed jqplot.pieRenderer.min.js to yjqplot.pieRenderer.min.js to make sure it loads after jquery.jqplot.min.js. I placed the CSS files from plugins in the client/css subdirectory to ensure that they load first.

Improving Sales Portal access security

So far, anyone who has the Sales Portal URL can see and even change the sales data. Although this is too permissive for any real-life use cases, the default prototype mode that is supported by Meteor is ideal for initial rapid evolution of your applications. During that phase you are likely modifying the interactions, the UI, and even application logic in rapid iterations — typically without involving any sensitive data. Having an open access model for the prototyping stage enables you to share the URL with collaborating developers and reviewing users to gather feedback.

The next natural step is to lock down the application by turning on Meteor's security features. The code for a substantially more secure (and therefore more realistic) version of the Sales Portal is in the download's sales subdirectory. Its added security features are:

Meteor Smart Packages

Smart Packages are modules of functionality that you can easily add to or remove from Meteor via the command line. A Smart Package can include server- and client-side code, UIs, APIs, and more. You see the example of `accounts`, `autopublish`, and `insecure` in this article. But you can

also add CoffeeScript support, URI routing, and many more features by adding Smart Packages. To see an up-to-date list of Smart Packages for the Meteor version that you're using, run the `meteor list` command.

- A user authentication system to allow only authorized user access to the portal
- Code that ensures that only an official owner of a region's sales data can modify that data
- Better source code organization to ensure that no server-side code is exposed to deployed clients

From this point on, all my references to the Sales Portal application are to the secure version that's in the sales directory.

Removing the ability for clients to modify server data

A good start for locking the application down is to prevent anyone from modifying the data. All you need to do in this case is to remove the `insecure` Smart Package with this command:

```
meteor remove insecure
```

The `insecure` Smart Package essentially tells the server not to check access rules before reading or changing data. This Smart Package is installed by default, allowing all access. After you remove it, no client can modify any server data. If you go back to one of the browser instances and try to modify any of the sales data, you'll notice that the application tries to change the data, but it quickly reverts, reflecting the denial of access from the server. (This is an example of latency compensation in action. The data is updated for a moment in the client, but once the authoritative server copy arrives, it overwrites the client's data.)

After you remove the `insecure` package, you must add access rules to explicitly allow access to specific pieces of data (by specific users). But there are no users yet. A user database and login authorization system must be added next.

Ensuring that only authorized users can view the sales data

Before you add the user authorization system, make sure that nobody can see the sales data. (Authorized users will be allowed to see it later.) Right now, even though they cannot modify the data, anyone can view it by going to the Sales Portal URL.

Remove the default `autopublish` Smart Package to prevent any Meteor collection data from being pushed from the server to the client — except data that is explicitly published by the server and subscribed to by the client:

```
meteor remove autopublish
```

If you visit the Sales Portal URL now, the regional sales data and pie chart aren't visible.

Adding user login via the `accounts` Smart Package

Meteor has available Smart Packages that make adding a user login and authorization system straightforward. The `accounts` Smart Package covers end-to-end workflow; it includes the front-end UI, back-end database, and client-to-server APIs that are required. You can add all of these features to Sales Portal with one command:

```
meteor add accounts-password accounts-ui
```

The `account-password` Smart Package supports user creation and login via familiar email-address-plus-password logins. The implementation uses the Secure Remote Password protocol (see Resources), and plain-text passwords are never sent between the client and server.

In addition to password-based login, you can also let your users sign in via Facebook, Twitter, Weibo, GitHub, Google, and Meetup — simply by adding one or more Smart Packages to your applications. Social network OAuth login support is less likely to be useful in enterprise intranet environments at this time, but these features are valuable for consumer-facing web or mobile applications.

Customizing user login UIs

If you want further control in styling the dialogs used in the UI widget, you can add the `accounts-ui-nostyle` package instead of the `accounts-ui` package. If you want to take over the UI completely, consult the Meteor documentation for a description of the API and dataflow used for the process (see Resources).

Drop-in UI for login

The `accounts-ui` package provides a prefabricated set of CSS-styled UI widgets (and supporting JavaScript code) to handle user login, new-user creation, and lost-password recovery. To add them, add the `{{loginButton}}` Handlebars helper. Listing 5 shows login added to the Sales Portal application in part of the sales/sales.html file:

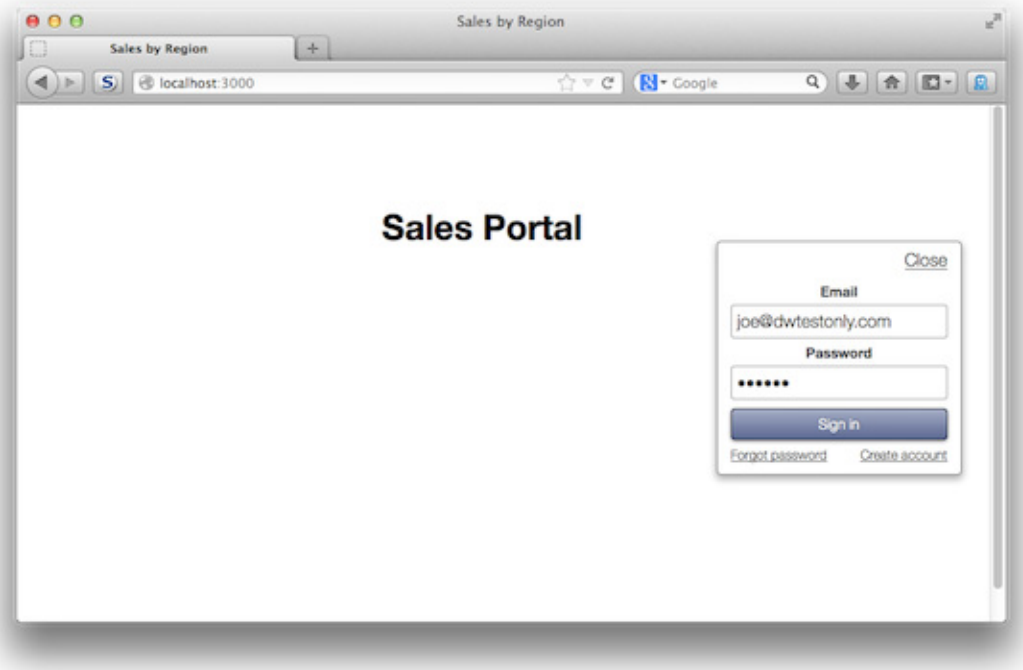
Listing 5. Adding a user login and authorization system

```
<body>
  <div id="title">
   <div class="header">
   <div class="span5">
     <h1 style="margin-bottom: 0px">Sales Portal</h1>
   </div>
   <div class="span5">  <div style="float: right">
    {{loginButtons align="right"}}  </div>
   </div>
   </div>
  </div>
```

The sales directory in the code download contains the completed Sales Portal application with user access control. Run this version to try logging in. Start a

browser instance, and notice that now there's a **Sign in** link in the upper-right corner. Click it, and you should see the dialog shown in Figure 5:

Figure 5. Sign-in dialog from the `accounts-ui` Smart Package (shown in Firefox)



The accounts Smart Package uses Meteor collections and publish-subscribe — the same facilities that are also available for you to use manually in your own code — to implement the user database. In the current version of Sales Portal, I've created two sets of user credentials, shown in Table 1, in the database:

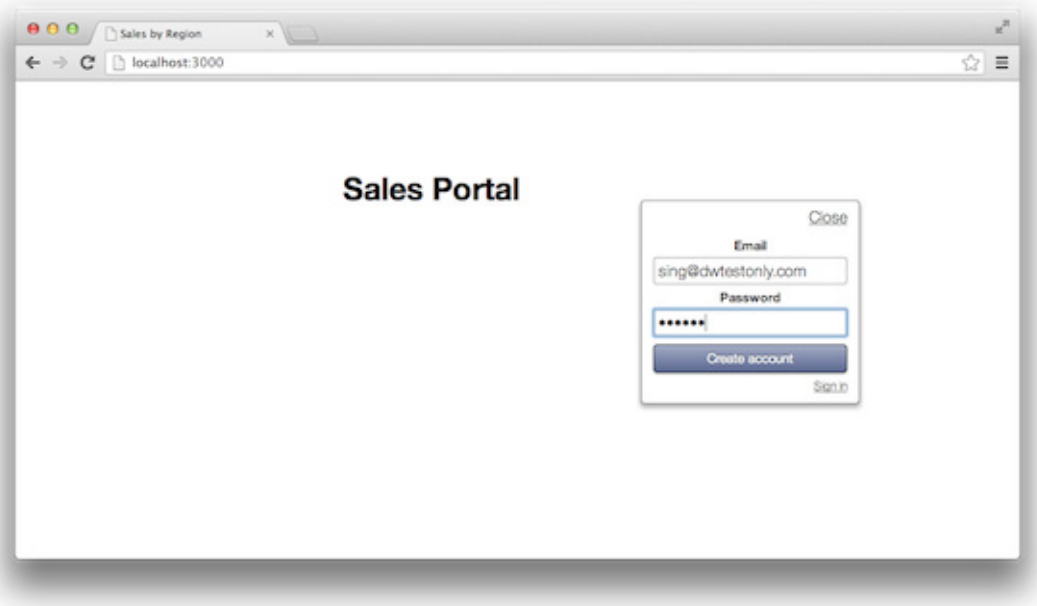
Table 1. Existing Sales Portal user credentials

Email	Password
joe@dwtestonly.com	abc123
sing@dwtestonly.com	abc123

Open two browser instances and log in with one credential each.

You can create more users by clicking the **Create account** link in the **Sign in** dialog. Figure 6 shows the create-new-user dialog that is a part of the `accounts-ui` Smart Package:

Figure 6. Dialog for creating new user accounts from the `accounts-ui` Smart Package (shown in Chrome)



Adding owner field to regional sales data

In the current version of Sales Portal, the initial database content has been modified. I used the server-side code in Listing 6 to seed the data:

Listing 6. Server-side data-seeding code

```
Sales2013.remove({});
Sales2013.insert({region:"US East", total: 2032333});
Sales2013.insert({region:"US Central", total: 150332, owner: joe._id});
Sales2013.insert({region:"US West", total: 1202412});
Sales2013.insert({region:"Asia Pacific", total: 701223});
```

A new `owner` field was added in Listing 6). In this case, the `owner` field contains the `userId` of the user (joe@dwtestonly.com) who owns the US Central region data. This field is used to restrict regional sales data updates only to joe@dwtestonly.com. You can obtain `userId` values by querying the `Meteor.users` collection.

Fine-grained selective server-data publishing

With the `autopublish` Smart Package removed, it is necessary to publish data from the server explicitly, and explicitly subscribe to it from the client.

For Sales Portal, the server publishes the `global_sales` collection using the code in Listing 7, which is part of the sales/server/sales.js file:

Listing 7. Selectively publishing data from the server

```
Meteor.publish("global_sales", function () {
      if (this.userId) {  // only visible to logged in users
      // do not include the owner field for client access
      return Sales2013.find({}, {fields: {"region": 1, "total":1 }});
      }
});
```

In Listing 7, note the use of `this.userId` to ensure that a valid user is logged in to the client-side session. When Meteor executes server code on behalf of a user, `this.userId` always contains the unique ID of the current logged-in user. If the current browser instance has no logged-in user, `this.userId` is null, and no data is published. Furthermore, in Listing 7 not all the fields of a regional sales data document (a *document* being essentially a record with a variable number of fields in a MongoDB instance) are returned in the collection sent to the client. Specifically, the document's `owner` field is hidden from the client. This is how, using a query, you can publish only a subset of the collection containing a subset of the fields, only to a client with authorized logged-in users. This technique is useful for ensuring that sensitive data fields in certain documents are inaccessible to client browsers.

Client-side data subscription

The Sales Portal client code explicitly subscribes to the server-published `global_sales` collection, as shown in Listing 8:

Listing 8. Client subscribing to a collection from server

```
Meteor.subscribe("global_sales");
Template.salesdata.dataset = function () {
  return Sales2013.find({});
};
```

Adding access rules to allow regional sales updates

With the `insecure` Smart Package removed, all users are effectively denied from updating the sales data. Assuming the different regional sales figures are owned by different users, an access rule can be added to allow update of the US Central data by joe@dwtestonly.com. Listing 9 shows this access rule in the server-side source file named model.js:

Listing 9. Server-side access rule allowing owner to update sales figure

```
Sales2013.allow({
  update: function (userId, sales, fields, modifier) {
    if (userId !== sales.owner)
      return false; // not the owner

    var allowed = ["total"];
    if (_.difference(fields, allowed).length)
      return false; // tried to write to forbidden field

    return true;
  },

});
```

The access-rule function for the `update` operation returns `true` if update is allowed and `false` if it is not allowed. The code in Listing 9 checks first to make sure that the user is the owner, and that only the `total` field is being modified.

Meteor hot code reload for iterative development

To save time during development and debugging, keep your browser pointed to the Meteor application even while you are changing code, CSS, or templates. Most of the time, Meteor's hot code reload feature detects the changes and pushes them to the client browsers.

Open Sales Portal and log in as joe@dwtestonly.com. Try to modify the US West data, and notice that it fails, then try to modify the US Central data. Because joe@dwtestonly.com is the owner if this data, you can update it.

Start another browser instance and log in as sing@dwtestonly.com. Try modifying any of the sales data and notice that it fails. Because sing@dwtestonly.com is not owner of any of the sales data, the server denies all modification requests from that user.

You can use the client-side `currentUser` function to avoid rendering the templates if the user is not logged in. Add the code in Listing 10 to the HTML file:

Listing 10. Eliminating attempts to render an empty template

```
<div id="container">
    <div id="salestable">
      {{#if currentUser}}
      {{> salesdata}}
      {{/if}}
    </div>
    <div id="chart">

    </div>
</div>
```

Now, start a new Sales Portal browser instance. Note that no data is visible. Log in as sing@dwtestonly.com, and notice that now you can see the data and pie chart. Now try to modify a field; again, you cannot modify it because you are not the owner.

Start another browser instance, log in as joe@dwtestonly.com, and note that the data is visible. Modify the US East figure, and notice that the pie chart updates. Confirm that the pie chart in the sing@dwtestonly.com session has also changed. Sign out of the two sessions and notice that the data now disappears.

Application deployment: Cloud and private

To simplify deployment, and to eliminate the need for you to set up your own servers while creating demos or experimenting with Meteor, the folks at Meteor came up with a single-command deployment of your application onto their cloud-hosted server. As of this writing, this service is free of charge. All you need to do is issue this command from your application directory:

```
meteor deploy applicationname.meteor.com
```

The application name must be unique because it becomes available (globally over the Internet) as http://*applicationname*.meteor.com. This article's applications are deployed at Meteor.com; see Resources for links to them.

If you want users to be able to access your application through your own company's domain — for example http://*applicationname*.*mycompany*.com/ — then you need to create a CNAME (alias) DNS record and point it to origin.meteor.com.

To host the application on your own server infrastructure, you need a server with node.js and MongoDB facilities preinstalled. You can create a deployable bundle of your application using the command:

```
meteor bundle applicationname.tgz
```

Fibers on node.js

Fibers is a relatively new addition to the node.js world. It enables nonpreemptive multitasking between separate logical flows (*fibers*). A fiber must explicitly yield execution before another fiber can execute. Because you control precisely the point at which execution yields, typically you don't need to protect shared state between fibers. The important feature of fiber is the notational convenience offered: It enables you to write code that looks sequential (such as one thread per logical flow) instead of the deeply nested callbacks typical of node.js programming. (See Resources for more information about node.js fibers.)

At the time of this writing, Meteor 0.6.3.1 requires that the self-deployed bundle be created on the same operating system as the eventual deployment system. This requirement is due to a native-compiled code dependency.

The Meteor server-side code runs on node.js *fibers*, providing a virtual environment where you can code as if one thread were handling each incoming request (with no sharable state). This approach can simplify the coding of server-side JavaScript logic.

Because the server-side deployable is a node.js application, you can customize the deployment topology to your own specific interoperation or scaling requirements.

Foto Share: A mobile photo-sharing service

Now that you've seen where a little design and planning with a bit of Meteor code can take you, you might be thinking up a startup project or two already. The next example will give you more ideas to apply while creating Meteor applications for mobile devices.

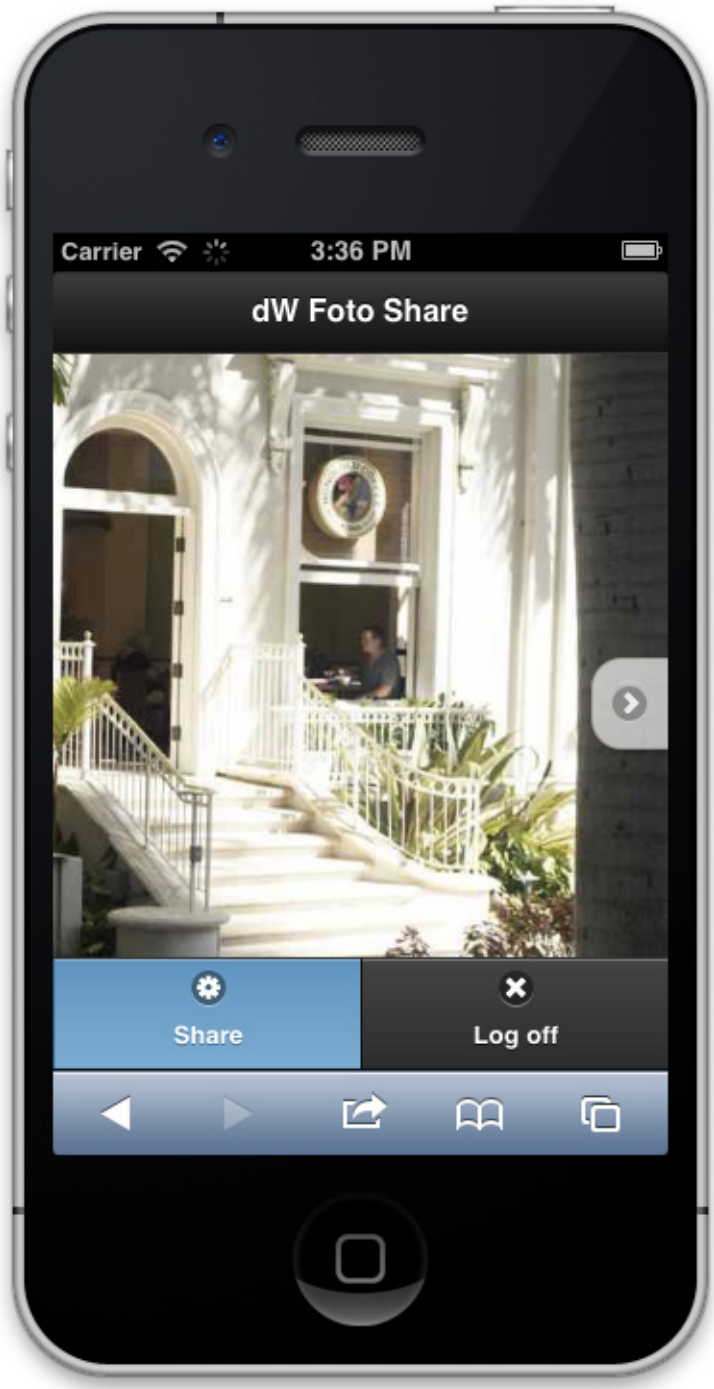
Meteor and mobile apps

Meteor features for creating mobile applications are still in the early stages of development. The entire Meteor project is a rapidly evolving work in

progress, and full mobile development support is targeted for after the 1.0 release.

Foto Share is a proof-of-concept photograph sharing service for mobile phone users. The use case is simple and direct: Users on their phone can browse through a collection of their photos, and share photos with their friends with one tap of the **Share** button. Figure 7 shows Foto Share running on an Apple iPhone:

Figure 7. Foto Share on Apple iPhone



Just as in the Sales Portal project, and for the same security reasons, the `autopublish` and `insecure` Smart Packages have been removed from Foto Share. And to implement password-based login, Foto Share has added the `accounts-ui` and `accounts-password` Smart Packages. The Foto Share application in the code download also has the same two users added as the Sales Portal.

To try out Foto Share, first run the application from the fotoshare directory in the code download. If you have two phones, point each one's browser to Foto Share. Otherwise, continue to use PC browsers. Sign in as sing@dwtestonly.com on one and joe@dwtestonly.com on the other (using the same password you used for Sales Portal). You can go through Sing's photos by swiping through them, or touch the overlay on either side. Each new photo slides into view. You discover that all of Sing's pictures are of scenes from Hawaii, and that Joe's are of Aztec and Mayan artifacts.

When you are ready to test share, choose a picture in Joe's collection and touch the **Share** button on the phone. On Sing's phone, Meteor reactively updates the subscribed collection, and Joe's shared picture is now viewable on Sing's phone.

Foto Share user login UI

Meteor doesn't (yet) have a mobile friendly `accounts-ui` Smart Package. It would be great to be able to "drop in" a customizable mobile login UI as you can for Meteor web applications. For Foto Share, I used the `accounts-ui` web UI. Figure 8 shows login dialog as presented on a mobile phone:

Figure 8. Foto Share login screen

Users typically don't mind logging in every time they reach a web application, but they expect any phone app to log in automatically on their behalf after they've provided the password for the first time. A possible solution for mobile Meteor apps is to wrap the client-side code in a native-app wrapper, such as the popular Apache Cordova platform (see Resources). You can then access the phone's own login profiles and

avoid asking the user to log in every time. Furthermore, you get direct access to the user's photograph collection stored in the phone's built-in gallery or album app.

Integrating with jQuery Mobile and pagination plugins

Taking advantage of the increasing compatibility of mobile browsers with HTML5, applications developed on the jQuery Mobile framework can typically run on the latest releases of major mobile OSs. The Sales Portal application has already shown how Meteor can integrate with jQuery and UI plugins such as jqPlot. As before, you need to be mindful of the loading order for CSS stylesheets and JavaScript files for the jQuery Mobile and pagination plugin libraries.

Because both jQuery Mobile and Meteor's LiveHTML (reactive rerendering) need to enhance and manipulate HTML elements through the browser's document object model (DOM), you must ensure that they occur in compatible order. In Foto Share, the page initialization by jQuery Mobile must be delayed until after Meteor's manipulations are complete. The aadelaybind.js file, containing the following code, is loaded before jQuery Mobile to delay its page initialization:

```
$(document).bind("mobileinit", function(){
  $.mobile.autoInitializePage = false;
});
```

After Meteor's LiveHTML completes its work, the actual page initialization is triggered by the `rendered` event helper of the `pages` template. The relevant part of this helper function is:

```
Template.pages.rendered = function() {
   ...
   $.mobile.initializePage();
   ...
};
```

Identifying reactive data in Foto Share

Conceptually, the most natural data collection to make reactive is the set of user's photos. Doing so would enable Meteor to update and rerender the list of photos whenever someone shared their photos. This is the approach taken in this proof of concept. For larger systems, depending on the back-end storage architecture for the images, you might want to make just the photo's metadata reactive instead of the images themselves.

Take a look at the server-side data-seeding code, shown in Listing 11, to get an idea of how the photos are stored:

Listing 11. Foto Share server-side data-seeding and collection-publishing code

```
Meteor.startup(function () {

      ...

      Fotos.remove({});
      Fotos.insert({name:"pic1", img: readPic('pic1.jpg'),
```

```
      owner: sing._id, shared:false});
     Fotos.insert({name:"pic2", img: readPic('pic2.jpg'),
      owner: sing._id, shared:false});
     Fotos.insert({name:"pic3", img: readPic('pic3.jpg'),
      owner: sing._id, shared:false});
     Fotos.insert({name:"pic4", img: readPic('pic4.jpg'),
      owner: joe._id, shared:false});
     Fotos.insert({name:"pic5", img: readPic('pic5.jpg'),
      owner: joe._id, shared:false});
     Fotos.insert({name:"pic6", img: readPic('pic6.jpg'),
      owner: joe._id, shared:false});

   Meteor.publish("photos", function () {
     if (this.userId) {  // only visible to logged in users
      return Fotos.find( {$or : [{owner: this.userId}, {shared: true}]},
  {fields: {"name": 1, "img":1 , "owner": 1}});
     }

  });
});
```

The server-published collection is `Fotos`. Each document in `Fotos` representing a photo has `name`, `img`, and `owner` fields. The `img` field is read in from a corresponding locally stored JPG file. Here again, notice that the data that a subscribing client receives contains only his or her own photos, plus any other photos that are shared by its owner. Here too, selective field filtering is used to remove the `shared` field from the Foto collections that clients receive. The `owner` field has not been filtered out because a client might want to show the name of the owner for a shared picture. Listing 12 shows the `readPic()` helper function. It uses synchronous fs to read the image into memory, and then encode the binary stream into base64 for storage into the `img` field. This format is convenient for displaying the photograph when it's retrieved on the client side.

Listing 12. Helper function to read JPG images for MongoDB storage

```
function readPic(infile)  {
    var fs = Npm.require('fs');
    var path = Npm.require('path');
    var base = path.resolve('.');
    var deployLoc = 'public/images/'
    var data = fs.readFileSync(path.join(base, deployLoc, infile));
    var tp = data.toString('base64');
    return  '' + tp;
}
```

When the images are reconstituted from the database, the template code takes advantage of *data URL* support in the `<IMG>` tag on most modern browsers (see Resources). Data URL support on the `SRC` attribute of an `<IMG>` tag enables the dynamic setting of the binary bits of an image through a base64-encoded string. Listing 13 shows part of the `photopage` template. In this template the base64-encoded `img` field from a photo is used to render the image in a jQuery Mobile page.

Listing 13. Setting an IMG tag's SRC attribute using data URL

```
<template name="photopage">
  <div data-role="page" id="p{{index}}">
    ...
    <div data-role="content" class="apic">
```

```
      <img src="{{img}}" />

      <ul data-role="pagination">
      {{#if indexIsZero}}
   <li class="ui-pagination-next"><a href="#p{{indexNext}}">Next</a></li>
      {{else}}
   <li class="ui-pagination-prev"><a href="#p{{indexPrev}}">Prev</a></li>
   <li id="x{{index}}" class="ui-pagination-next">
     <a href="#p{{indexNext}}">Next</a></li>
      {{/if}}
      </ul>

    </div> <!-- /content -->
    ...
  </div>  <!-- /page -->
</template>
```

Meteor Remote Methods: Custom RPCs made simple

Even though the `insecure` Smart Package has been removed, no access rule has been created in Foto Share. Without access rules, no client can update data via Minimongo. But when the **Share** button is tapped, the `share` field of the photo must have been updated somehow. How? The answer is Meteor Methods.

Meteor Methods is a remote procedure call (RPC) mechanism. You can create RPC calls from the client to the server in two simple steps:

1. Define a JavaScript function on the server side.
2. Use `Meteor.call()` to invoke the server function remotely, optionally passing arguments.

Meteor takes care of all the endpoint setup, takedown, and data-marshalling chores in between.

When you tap the **Share** button in Foto Share, the client calls a Meteor Remote Method on the server named `shareThisPhoto` and passes the photo's ID as an argument. On the server side, the code first checks to see if the caller is the owner of the photo, and it updates the photos `shared` field only if the owner called the method. Listing 14 shows the server-side `shareThisPhoto` code:

Listing 14. Meteor Remote Method on the server side to update a photo's `shared` field

```
Meteor.methods({
  shareThisPhoto: function (photoId) {
    console.log('called shareThisPhoto');
    console.log(photoId);
    var curPhoto = Fotos.findOne({_id: photoId});
    if (this.userId !== curPhoto.owner)  {
      return "Cannot share this photo.";
    } else {
      Fotos.update({_id: photoId}, {$set :{shared: true}});
      return "Photo shared!";
    }

  },
});
```

Listing 15 shows the client-side code that invokes the remote method when you tap the **Share** button:

Listing 15. Client-side code to invoke the remote method when the Share button is tapped

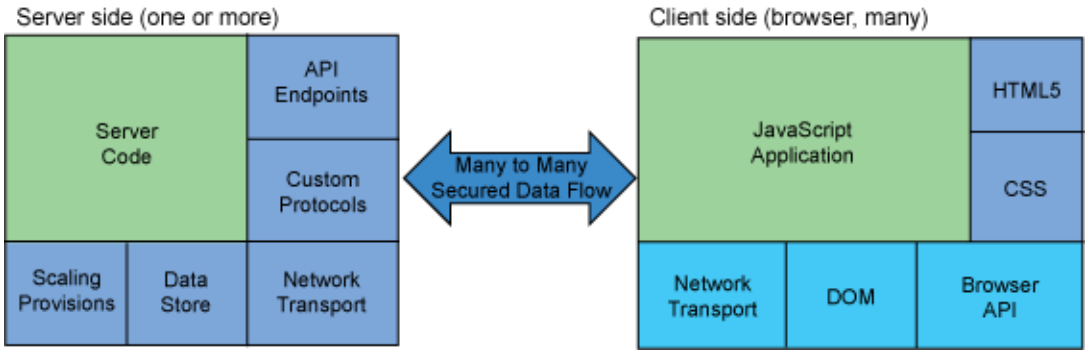
```
Template.photopage.events({
  'click .fs-logoff': function () {
    Meteor.logout(function() {
      location.reload();
    });
  },

  'click .fs-share': function() {
      Meteor.call('shareThisPhoto', this._id, function (error, retval) {
 console.log(retval);
      });
  }
});
```

I've chosen the RPC approach to demonstrate Meteor Remote Methods. You can instead define an access rule to allow an owner to update the `shared` field. In that case, you then must locally update the `shared` field of the photo when the user taps the Share button. Meteor's Minimongo pushes the update to the server, then to all other subscribed clients.

An architecture for modern web applications

Now that you've worked through this article's sample applications, it should make sense that Meteor is designed for web applications with the specific architecture represented in Figure 9:

Figure 9. Rich-client interactive web application architecture



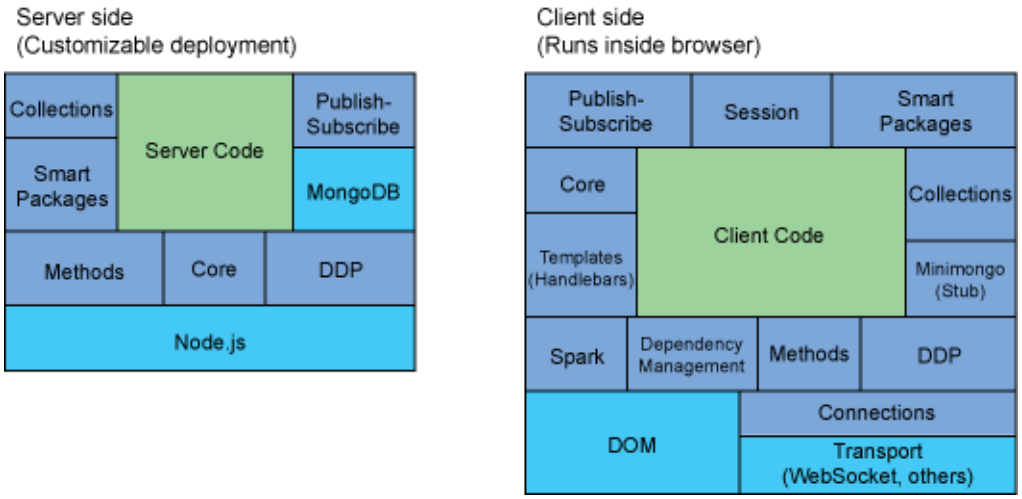
These types of applications typically consist of a highly interactive single-page UI. Users typically don't experience new page loads; instead, a portion of the displayed page updates instantaneously in response to user interaction, with few or no network round-trip delays. The single-page interface in no way restricts the application because portions of the page can be updated in an infinite variety of ways. This is reminiscent of a stand-alone desktop application such as a word processor or spreadsheet.

These types of applications typically load JavaScript application code on the client browser. This code manages the interaction with the user by dynamically manipulating the browser's DOM, modifying CSS styling, generating new HTML elements/code/styling, and leveraging other browser-provided APIs. All interfacing with the user is controlled by the client-side code; no additional HTML or styling is loaded via the network except for the initial application load. The same code also shuttles data back and forth between the client and the server(s) to implement application features. The browser essentially loads and runs a rich-client (sometimes called fat-client) application written in JavaScript.

On the server side, endpoints are set up to source and sync data from the client securely. Legacy back ends can have RPCs, XML-based web services, RESTful services, or other JSON-style RPC calls. Modern back ends are likely to serve proprietary protocols that are designed to be efficient for the data on the wire, resilient to occasional disconnection, supportive of a variety of current transports, and topologically scalable.

Figure 10 shows the major components internal to Meteor as of release 0.6.3.1:

Figure 10. Meteor internal components



As you already know, DDP offers bi-directional dataflow between client and server instances, and Minimongo is a smart stub with local data cache on the client side that offers the familiar MongoDB query API to client-side code. Spark is the live HTML engine that works along with (Handlebars) templates and dependency management to provide Meteor's reactive rerendering feature. (See Resources for a link to the official Meteor documentation, where you'll find a detailed explanation of the Meteor components.)

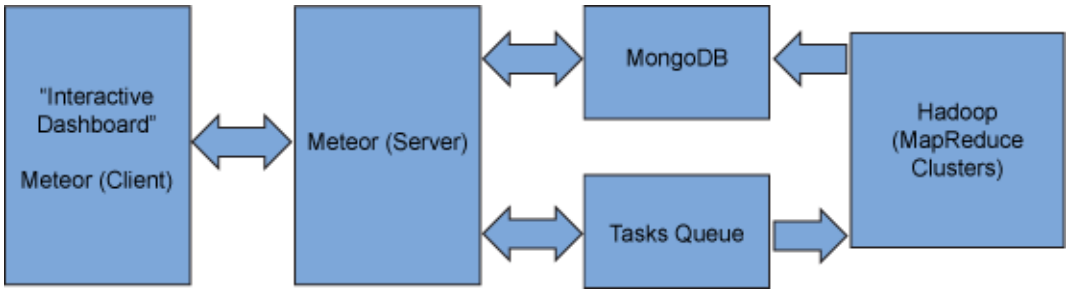
Potential big data application architecture

Meteor's one page, real-time, highly interactive paradigm is almost custom-tailored to fit a certain class of problem. One aspect of visualizing big data is the need

for interactive dashboards that update as soon as results become available. The dashboard might also be used to queue MapReduce jobs and monitor their real-time progress.

A reactive Meteor application can potentially provide an interactive interface into terabytes or petabytes of data — just as a spreadsheet can offer interactive consolidation, summary, drill-down, or customized views into a relatively small set of data. Figure 11 shows the architecture for such a system:

Figure 11. Architecture of a Meteor dashboard application for big data



In Figure 11, the client-side Meteor code offers a reactive dashboard view into the big data repository. User interactions generate customized MapReduce tasks that are queued, possibly via Meteor Methods, on the server-side for execution on Hadoop clusters. As the execution of tasks completes, results are consolidated into the MongoDB instance via Meteor. The Meteor server's publish-subscribe component detects data changes and pushes updated summary data out to the subscribed client(s).

Conclusion

Meteor interview

Learn more about the Meteor project's background and future plans in an interview that Sing Li conducted with Meteor co-founder Matt DeBergalis.

The Meteor team is working nonstop to level the playing field for everyone who wants to create single-page, highly interactive, rich-client web applications. Most users prefer such applications over the legacy page-at-a-time style. Many of the major web-based mail and office services — including Google Gmail, Microsoft Hotmail, and Yahoo Mail — use this application architecture. We don't see many instances of these kinds of web applications other than ones developed by the biggest tech companies or most-well-funded startups because they are difficult to build, requiring significant effort and resources. Meteor's stated core mission is to provide a base on which this class of application can be built easily.

Meteor's entrance on the scene makes this is a great time to revisit all those long-forgotten spreadsheet application ideas you have stashed away, and see if converting them into a public-facing service might be an appealing venture. If you are

already creating or maintaining web applications, Meteor might prove to be the most potent tool in your ever-expanding array of options.

Downloads

Description	Name	Size	Download method
Sales Portal and Foto Share code	instmeteordl.zip	367KB	HTTP

Information about download methods

Resources

Learn

- Meteor: Explore the Meteor website.
- Meteor documentation: Check out the official Meteor documentation for the latest features and updates.
- "Develop easy real-time websites with Meteor" (David Berube, developerWorks, January 2013): This article is a step-by-step guide to installing Meteor and getting started with Meteor development.
- sales_nologin, sales, and Foto Share: Meteor currently supports Mac OS X and Linux on i686/x64 platforms. Windows official support is coming around release 1.0. Meanwhile, Windows users can run the deployed versions of this article's sample applications. (Be aware that multiple users might be accessing them at the same time.)
- Meteor on Twitter: Follow Meteor to find out about Meteor learning events and opportunities around the world.
- MongoDB: The MongoDB NoSQL database stores JSON-style documents and has a succinct query syntax that is used extensively on both client and server Meteor application code.
- "Discover MongoDB" (Andrew Glover, developerWorks, December 2012): Learn more about MongoDB in this knowledge path.
- jQuery: Discover everything related to the jQuery framework, including a large list of plugins for different jQuery versions.
- jqPlot jQuery plugin: Much more than just pie and bar charts, find out what jqPlot can do for you.
- Jeditable jQuery plugin: For fast and simple in-place editing support in your jQuery-based UI, Jeditable does the job well.
- jQuery UI: This plugin offers a touch-friendly interface for transitioning between multiple jQuery Mobile pages.
- Fibers for Node: The Fibers module adds nonpreemptive multitasking to node.js and is a foundational piece for server-side deployment of Meteor.
- Secure Remote Password protocol: Explore the SRP protocol, gives Meteor's `accounts` Smart Package a means of authenticating a user without ever sending cleartext password over the wire.
- IETF RFC-2397: The "data" URL scheme is supported to some extent by most modern desktop and mobile browsers.

Get products and technologies

- Meteor: Meteor is available for download on GitHub.
- jQuery Mobile: Start developing web applications that can run on multiple modern mobile devices.
- Apache Cordova: You can use Cordova to create native mobile applications wraps for JavaScript, HTML, and CSS code.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Sing Li

Sing Li has been a developerWorks author since the site's inception, writing articles and tutorials that cover a variety of web and Java topics. He has more than two decades of system engineering experience, starting with embedded systems, crossing over to scalable enterprise systems, and now back full circle with web-scale mobile-enabled services and "Internet of things" ecosystems.